



Tutorial 5: GDB, Make

CS 104

Spring, 2024-25

TA: Mohana Evuri

Credits: Saksham Rathi (2023 - TA)

Compiling C and C++ files through terminal

- To compile C and C++ files through the terminal, we typically use the `gcc` (GNU Compiler Collection) for C files and `g++` for C++ files.
- `gcc -o output_file input_file.c`: `gcc` is the compiler. `-o` specifies the executable file name, `input_file.c` is the file to be compiled.
- `g++ -o output_file input_file.cpp`: `g++` is the compiler. `-o` specifies the executable file name, `input_file.cpp` is the file to be compiled.
- Multiple files can be compiled into a single executable using `g++ -o output.o file1.cpp file2.cpp file3.cpp`
- Multiple object files can be linked together using: `g++ -o linked_executable file1.o file2.o file3.o`

Common Flags



1. `-Wall`: Enable most warnings.
2. `-std=c++20`: Use C++20 standard.
3. `-g`: Generate debugging information.
4. `-O2`: Optimize code.
5. `-Werror`: Treats all warnings as errors.
6. `-Wno-()`: Suppresses all the warnings of the type `()`
 - Eg: `-Wno-unused-variable`



Make

Introduction to Make



- Makefiles are generally used to help decide which parts of a large program need to be recompiled.
- So, basically it simplifies project management. For large project with minimal changes, we need not compile the entire project again (it might even take few hours to do so!). So, make automatically compiles those parts where we have made some changes.
- It also gives us a brief overview of the project structure and dependencies. So, if any file's dependencies are changed, that file will be recompiled.
- Make can also be used beyond compilation too, when we need a series of instructions to run depending on what files have changed.
- **Cmake**: Open-source cross platform family to tools to build software.

Installation of Make



Linux - Debian: `sudo apt install make`

Linux - Other distros: Use your distro's Package manager.

macOS: (With homebrew) Open terminal and type: `brew install make`

Windows: Use **WSL**. If you REALLY want to install (not recommended, as in the future courses: **DSA, OS** - you would definitely require WSL):

[How to install and use "make" in Windows? - Stack Overflow](#)

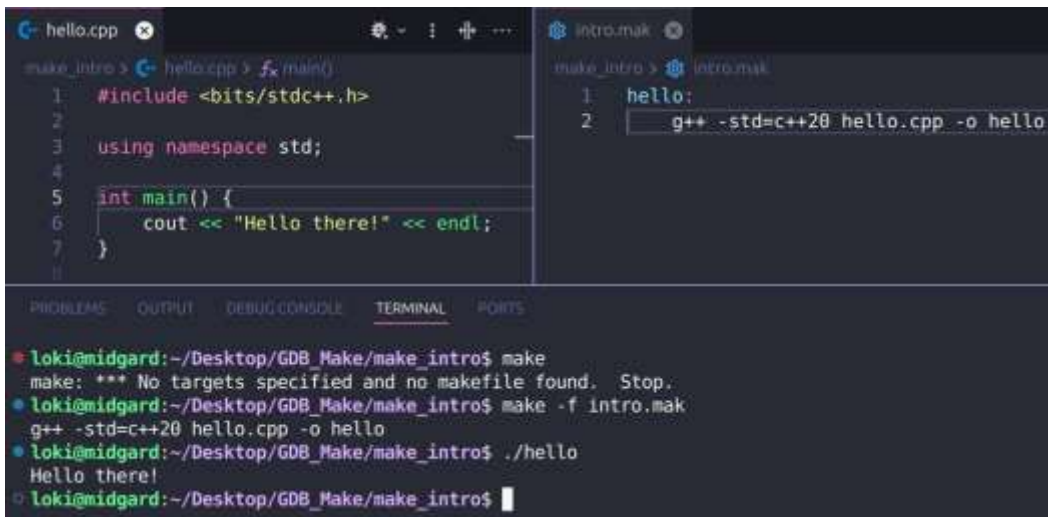
Syntax of Makefiles



```
targets: prerequisites  
    command  
    command  
    command
```

- Targets are file names which will be ready after the commands get executed. They are generally executables or binary files. If we have more than one file targets, separate them by spaces.
- Prerequisites are the dependencies for the target. They need to exist for the commands to run. If we have more than one file dependencies, separate them by spaces.
- Commands are the actions that need to be carried out. They are only carried out if the prerequisites are changed or the targets do not exist. These commands need to start with tab not with spaces. Make relies on timestamps, so the commands will run if targets have older timestamps than prerequisites.

Naming of Makefiles



```
hello.cpp
1 #include <bits/stdc++.h>
2
3 using namespace std;
4
5 int main() {
6     cout << "Hello there!" << endl;
7 }
8

intro.mak
1 hello:
2     g++ -std=c++20 hello.cpp -o hello

loki@midgard:~/Desktop/GDB_Make/make_intro$ make
make: *** No targets specified and no makefile found. Stop.
loki@midgard:~/Desktop/GDB_Make/make_intro$ make -f intro.mak
g++ -std=c++20 hello.cpp -o hello
loki@midgard:~/Desktop/GDB_Make/make_intro$ ./hello
Hello there!
loki@midgard:~/Desktop/GDB_Make/make_intro$
```

- `make` command looks for the presence of a Makefile in the same directory.
- It tries the following names in order:
 1. `GNUmakefile` (For makefiles specific to GNU Make)
 2. `makefile`
 3. `Makefile` (Most preferred)
- If you wish to use a different name, then just use `make -f`

Targets

```
all: one two three

one:
    touch one
two:
    touch two
three:
    touch three

clean:
    rm -f one two three
```

- Make by default runs the **first** target.
- If you wish to make multiple targets and run all of them simultaneously, then you can use all target. Since, this is the first rule listed, typing “**make**” on the terminal without specifying any target will run this.
- **clean** is often used as a target that removes output of other targets. It is a target that is not the first, so will not run unless you explicitly type “**make clean**” on the terminal.

Phony Target

```
some_file:
    touch some_file
    touch clean

.PHONY: clean
clean:
    rm -f some_file
    rm -f clean
```

- Adding a **.PHONY** to a target will prevent make from confusing the target with a file name.
- In the example shown, the target “**clean**” will still run, even if the file “**clean**” is created.
- Additionally, phony targets have names that are rarely file names and in practice, many people skip this.

Variables

```
files := file1 file2
some_file: $(files)
    echo "Look at this variable: " $(files)
    touch some_file
```

```
x := dude

all:
    echo $(x)
    echo ${x}

# Bad practice, but works
echo $x
```

```
MESSAGE = "Hello, World!"
all:
    echo $(MESSAGE)
    echo "$(MESSAGE)"
```

```
) make
echo "Hello, World!"
Hello, World!
echo "'Hello, World!'"
Hello, World!
```

- Variables can only be strings.
- Typically, “:=” is used, but “=” also works.
- Variables are referenced as `$()` or `${}`. Don’t confuse with the shell command substitution. We use `$$()` for that in make.
- Single or double quotes have no meaning to make, they are simply characters assigned to the variables.

Automatic Variables and WildCards

```
hey: one two
# Outputs "hey", since this is the target name
echo $@

# Outputs all prerequisites newer than the target
echo $?

# Outputs all prerequisites
echo $^

touch hey
```

M Makefile

```
1 %.o: %.c
2 | echo "hello"
```

- `$@`, `$?`, `$^` and `$<` are the automatic variables, see the example on how to use them.
- `$<` - The first prerequisite (dependency).
- Both `*` and `%` are the wildcards used in make.
- `*` searches for filenames in our system. For example `*.c` will match all filenames with the extension `.c`.
- `%` is used in various cases, here for all `.o` files, the corresponding `.c` file is searched for and the “hello” is printed.
- Overall, `%` is used for pattern matching and `*` is used for variable expansion and shell commands.

Implicit Rules



- Make loves C compilation, so it has some inbuilt automatic rules to make our work easy. (At times, things get confusing using these rules!)
- The important variables used by implicit (only for **Linux**) rules are:
 1. **CC**: Program for compiling C programs, default: **cc**.
 2. **CXX**: Program for compiling C++ programs, default: **g++**.
 3. **CFLAGS**: Extra flags to give to the C compiler.
 4. **CXXFLAGS**: Extra flags to the C++ compiler.
 5. **CPPFLAGS**: Extra flags to give to the C preprocessor.
 6. **LDFLAGS**: Extra flags to give to compilers when they are supposed to invoke the linker.

`n.o` is made automatically from `n.cc` or `n.cpp` with a command of the form:
`$(CXX) -c $(CPPFLAGS) $(CXXFLAGS) $^ -o $@.`



GDB

GNU DeBugger

Debugging



Process of finding compile and run-time errors is called debugging.

- Compile time errors:
 1. Missing semicolons. (after python practice!)
 2. Wrong variable names.
 3. Out of scope variables.
 4. Assigning strings to integer variables. (Rare?)
 5. Didn't pass arguments to functions.
 6. Some others left? GDB can handle all of them!
- Run-time errors:
 1. Illegal use of pointers.
 2. Freeing memory that was already freed.
 3. Array index out of bounds.
 4. Dereferencing memory that has not been allocated.

Alternatives to GDB

```
for (int i = 0; i < numElements; ++i) {  
    #ifdef DEBUG  
    cerr << "i: " << i << endl; // Only included if DEBUG is defined  
    #endif  
    cout << myArray[i] << endl; // Always included  
}  
#ifdef DEBUG  
cerr << "Exited loop" << endl; // Only included if DEBUG is defined  
#endif
```

```
unsigned long gcd(unsigned long m, unsigned long n) {  
    assert(m >= n);  
  
    while (n != 0) {  
        unsigned long r{m % n};  
        m = n;  
        n = r;  
    }  
  
    return m;  
}
```

- Add output statements to the code. (My favourite!) Better to use endl instead of “\n”, because “endl” causes a flush operation, which means any data to be printed is directly sent to the terminal. Usage: `cout << “debugging is necessary ” << x << endl;`.
- You can even set compile time symbol `DEBUG`. `g++ -g -c -DDEBUG myProgram.cpp`. This will make sure that the `DEBUG` sections will be included while executing the code.
- Assertions: “Programming with defense”. Include `<assert.h>` in C++ to use this. Program fails with an informational message whenever the assert condition fails. If compile time symbol “`NDEBUG`” is used, then assertion is commented.
- VS Code Inbuilt Debugger (Install Extensions).

How to start GDB?



- Compile your program with the `-g` compile flag. This helps in debugging.
- Compile: `g++ -g -o try try.cpp`.
- Debugging: `gdb try`.
- Starts the debugger for this program, but the program does not start running.
- `gdb`: Starts the debugger but without any file.
- `gdb --help`: Shows the possible command line options.
- `quit`: Exit `gdb`, `q` will also work.

Breakpoints



- To stop the program at different points in its execution, we use breakpoints.
- To specify a breakpoint upon entry to a function, we use: “`break function`” (stops just before the start of the function), “`break bug.c:function`” (stops before the function present in a particular file), “`b function`” (b = shortcut for break).
- “`break 26`” will set a breakpoint at line number 26 of the file. Use filenames when we have more than one file.
- To list current breakpoints, type “`info breakpoints`”. Breakpoints will be given some numbers(id’s). They can be deleted using “`delete <id>`”.
- Breakpoints can also made to be triggered only when certain conditions are met. “`break <line/function> if <condition>`”.
- `break +offset`: will set a breakpoint after offset lines from current spot.
- `tbreak`: temporary break, remove when reached.
- `rbreak <regex>`: break on all functions matching regex.
- `ignore <n> <count>`: ignore breakpoint n count times.

Execution Control



- `(gdb) run/r`: Starts running the program.
- `continue/c`: Continue running after a breakpoint. (run starts from the beginning of the program, whereas this starts from the breakpoint itself!).
- `next/n`: Execute next line stepping over function calls.
- `step/s`: Execute next line stepping into function calls, if any.
- With all these three, we can even specify the number of times, they are supposed to be run.
- `(gdb) help`: Lists the possible topics we can ask for.

Program Stack



- `backtrace/bt`: Prints the current contents of the **stack**. Also prints the arguments to the function calls.
- `up`: Helps move one step up in the stack i.e. from a function to its caller.
- `down`: Helps move one step down in the stack, i.e. from a function to its callee.
- `info args`: Print the arguments of the current function.
- `info locals`: Print the local variables of the current function.

Printing Variables



- `display <variable>`: Print the values of the variable, every time the program stops at a breakpoint.
- `print <variable>`: Prints the value of the variable at that particular breakpoint. (Shortcut: `p`).
- We can also use `print` to evaluate expressions/function calls, reassign variables and more... Example: `print strlen(text)`.
- `list/l`: Shows next ten lines of the source code.
- `show dir`: Shows current source path.
- `shell <command>`: Runs the command as if it is in a shell (`ls`, `cat`, etc...).



Thank You ☐ !!!